Dangerous constructs

Arjen Markus (12 november 2003) In response to a recent discussion on the c.l.t. about a problem that arose in the context of regular expressions, I have started this page. Its sole purpose: *document dangerous constructs in Tcl*

Using the subst command on arbitrary data

```
set a "Hello,"
set b "world!"
set string "\$a \$b"
puts [subst $string]
```

gives:

```
 Hello, world!
```

but:

```
set string "\[exit\]"
puts [subst $string]
```

stops your program!

The subst command allows you to suppress the execution of commands:

```
puts [subst -nocommands $string]
```

gives:

```
  [exit]
```

DKF: The way to protect against such things is to use safe interpreters. That doesn't stop you from making programming mistakes, but it does mean the leaks occur in a sandbox.

DKF: Also, [subst -nocommands] doesn't protect against:

```
set string "foo,\$a(\[exit\]),$b,bar"
```

since that's principally a variable substitution.

Unexpected 0ctals

Remember that if Tcl is expecting an integer, and the value begins with a 0 (zero), it will interpret it as an octal number. This bug is triggered most often when trying to do arithmetic on date and time values obtained from [clock format]. This is especially

dangerous since you can (e.g.) write a program in January that works just fine for months, then crashes in August and September.

Solution : The scan page recommends:

```
scan $possiblyZeroedDecimal %d cleanInteger
```

Anything goes (in a switch)

RS: A simple error that will appear only at runtime is **not protecting a switch command with --**:

```
switch $input {…}
```

The error will occur if $input starts with a minus (-) sign. So best always use

```
switch -- $input {...}
```

LV There are a number of other tcl commands which also support **--** ; if the command supports it, and you are using *random* input from users or input files, you probably should use it.

DKF: The exact semantics of switch were changed (in 8.6?) to ease this. Now, if called with only two arguments, switch assumes the first is a value to switch on and the second is a collection of patterns and bodies. (The compiler then issues a jump table to implement it, which is rather fast.)

Expecting that only one other party can predict a socket

TV opening any server socket, expecting a certain other party to connect. For instance a file transfer à la ftp where a control connection triggers a file transfer over a separate socket pair.

DKF: Fixing this requires that you do an identification exchange after opening the socket. And possibly that you use tls::socket to open the socket in the first place. FTP is unfixable without going to major effort (FTPS is rarely deployed, and sftp works in a totally different way under the covers).

Unbraced expressions

What's *dangerous* about **unbraced expr**?

Short answer:

```
# Uh-oh; what if it's "exec rm -rf ..." rather than "exec touch ..."?
set a {[exec touch /tmp/77]}
set b {[exec touch /tmp/78]}
catch {expr $a + 4}
catch {expr {$b + 4}}
```

DKF: Protection is two-fold. Brace those expressions unless you're really certain you know better, and use safe interpreters (see subst discussion above).

Verify that something in a variable is an integer by using:

```
incr theVar 0
```

Verify that the value in a variable is numeric by using:

```
expr {$theVar + 0.0}
```

(Note: the expression is *braced* so it is safe.)

Verify that something is a list or dictionary by using:

```
list $thePossibleList
dict size $thePossibleDict
```

Defaults for list searching

AM Here is another one:

```
set list {A B C D E}
set elem B
set a [lsearch $list $elem]
```

This will work nicely, unless you have a list like this:

```
set list {A B ? D E}
```

and you look for "?". This will in fact return 0, not 2!

You should use:

```
set a [lsearch -exact $list $elem]
```

as lsearch uses glob patterns by default

DKF: Sometimes in 8.5, it is better to say:

```
if {$elem in $list} {
    ...
}
```

Comparisons also handle strings

RS once had to debug this:

```
proc foo max {
    for {set i 0} {$i<=$max} {incr i} {
        #do something with $i
    }
}
```

Harmless enough. But if called with a nonnumeric argument *max*, say *foo bar*, it will never terminate, because any integer value of *i*, in string comparison that is forced in this case, is less than *bar*. Simple remedy: add

```
incr max 0
```

before the for loop - that will throw an error *expected integer but got bar* which is better to understand and fix than an endless loop somewhere deep in the code....

See also

- Frequently Made Mistakes, FMM
- Update considered harmful

Updated 2014-06-07 20:40:34